AHDI 3.00 Release Notes

April 18, 1990

Atari Corporation
1196 Borregas Avenue
Sunnyvale, CA 94086

AHDI 3.00 Release Notes were printed in the United States of America.

Second Edition: April 18, 1990

This document was produced entirely with Microsoft Write, an Atari Mega 4 computer, and an Atari SLM804 laser printer.

Hard Disk Partitioning

AHDI 3.00 adds support for hard disks with more than four partitions and partitions of size greater than or equal to 16 Mb. There are still only four 12-byte structures (called p?_info) to describe partition information in the first sector (physical sector #0) on a hard disk. Physical sector #0 is called the root sector.

The root sector (physical sector #0) on a hard disk contains this information:

```
                                    Offset
+----------------------------+      ($1c2)
|          hd_siz            |
+----------------------------+      ($1c6)
|          p0_flg            |
|          p0_id             |
|          p0_st             |
|          p0_siz            |
+----------------------------+      ($1d2)
|          p1_flg            |
|          p1_id             |
|          p1_st             |
|          p1_siz            |
+----------------------------+      ($1de)
|          p2_flg            |
|          p2_id             |
|          p2_st             |
|          p2_siz            |
+----------------------------+      ($1ea)
|          p3_flg            |
|          p3_id             |
|          p3_st             |
|          p3_siz            |
+----------------------------+      ($1f6)
|          bsl_st            |
+----------------------------+      ($1fa)
|          bsl_cnt           |
+----------------------------+      ($1fe)
|          (reserved)        |
+----------------------------+      ($200)
```

Figure 1

hd_siz	A 68000 format long word that contains the total size of the disk, in number of physical (512-byte) sectors.
bsl_st	A 68000 format long word that specifies the offset to the beginning of the bad sector list from the beginning of the entire hard disk, in number of physical (512-byte) sectors. (Typically the bad sector list will be located at the beginning of the device right after the root sector.)
bsl_cnt	A 68000 format long word that specifies the size of the bad sector list, in number of physical (512-byte) sectors.

There are two kinds of partitions, *standard* partitions and *extended* partitions. A *standard* partition can be a *regular* partition (that is, a partition whose size is < 16Mb), or a *big* partition (that is, a partition whose size is >= 16Mb). The first sector in a standard partition is a boot sector (which, on the ST, will contain a BIOS Parameter Block). For more information about big partitions, refer to BIG GEMDOS PARTITIONS. An *extended* partition is a special kind of partition which itself is subdivided into standard partitions. For more information about extended partitions, refer to EXTENDED GEMDOS PARTITIONS.

A hard disk may contain up to four standard partitions, or up to three standard partitions and one extended partition.

```
+---------------------------+
|       root sector         |
+---------------------------+
+---------------------------+
|       bad sector list     |
+---------------------------+
+---------------------------+
|       partition 0         |
+---------------------------+
+---------------------------+
|       partition 1         |
|       (optional)          |
+---------------------------+
+---------------------------+
|       partition 2         |
|       (optional)          |
+---------------------------+
+---------------------------+
|       partition 3         |
|       (optional)          |
+---------------------------+
```

Figure 2

Each partition (standard or extended) is described by a 12-byte partition information structure (p?_info where ? = 0, 1, 2, 3):

```
p?_info ------> +------------------------+     (+0)
                |        p?_flg          |
                +------------------------+     (+1)
                |        p?_id           |
                +------------------------+     (+4)
                |        p?_st           |
                +------------------------+     (+8)
                |        p?_siz          |
                +------------------------+     (+12)
```

Figure 3

p?_flg A 1-byte field that indicates the state of a partition.

Bit 0	When set, the partition exists. When not set, the partition does not exist.
Bit 1-6	These bits are reserved for future use.
Bit 7	When set, the partition is bootable. When not set, the partition is not bootable.

The BIOS will boot the first partition that has bit 7 set in this byte.

p?_id A 3-byte field that identifies the partition. The field may contain the three ASCII characters:
"GEM" - for regular (< 16Mb) GEMDOS partitions
"BGM" - for big (>= 16Mb) GEMDOS partitions
"XGM" - for extended GEMDOS partitions

p?_st A 68000 format long word that specifies the offset to the beginning of the partition from the beginning of the entire hard disk, in number of physical (512-byte) sectors.

p?_siz A 68000 format long word that specifies the size of the partition, in number of physical (512-byte) sectors.

Extended GEMDOS Partitions

The extended GEMDOS partition enables a hard disk to contain more than 4 partitions. Only one of the 4 partition information structures (p?_info as defined above) can contain an extended partition. The extended partition is identified by the ASCII characters "XGM" in the 3-byte p?_id field of a p?_info structure in the root sector. For more information about the structure of the extended GEMDOS partition, refer to INSIDE THE EXTENDED GEMDOS PARTITION.

Since an extended partition is not bootable, it *must* be preceeded by at least one standard partition, so the hard disk can be made bootable. This requirement makes it impossible for partition 0 to be an extended partition. A partitioning utility (e.g. HDX) should only create an extended partition if at least one preceeding standard partition already exists. A utility that installs a bootable driver onto the hard disk (e.g. HINSTALL) should never mark an extended partition as bootable.

Inside the Extended GEMDOS Partition

The extended GEMDOS partition is a partition which is subdivided into smaller ones. Each subdivision consists of an extended root sector (a 512-byte sector), and a standard partition. Conceptually, each subdivision is like a stand-alone hard disk with only one partition on it. These subdivisions are "linked" together by a pointer in the extended root sector.

The layout of an extended root sector resembles that of the root sector (refer to Figure 1), except that fields like hd_siz, bsl_st and bsl_cnt are not applicable in an extended root sector. Only two of the four p?_info structures would be used, but not necessarily the first two. One of the p?_info structures is used to describe the standard partition in the current subdivision, the other one provides a link to the next subdivision. The link should occupy a p?_info structure that follows the p?_info structure for the description of the standard partition. The other two unused p?_info structures should be filled with zeroes. Refer to Figure 3 for the layout of a p?_info structure.

For the standard partition description, the definitions of the fields in a p?_info structure are:

p?_flg	A 1-byte field that is used as a bit-vector of flags. Currently only bit 0 is being used; when set it indicates that this p?_info structure is being used. The remaining bits are reserved for future use.
p?_id	A 3-byte field that identifies the partition. The field *must* contain the three ASCII characters: "GEM" - for regular (< 16Mb) GEMDOS partitions "BGM" - for big (>= 16Mb) GEMDOS partitions
p?_st	A 68000 format long word that specifies the offset to the beginning of the standard partition from the beginning of the extended root sector that this structure resides in, in number of physical (512-byte) sectors.
p?_siz	A 68000 format long word that specifies the size of the standard partition in number of physical (512-byte) sectors.

For the link to the next subdivision, the definitions of the fields in the p?_info structure are:

p?_flg	A 1-byte field that is used as a bit-vector of flags. Currently only bit 0 is being used; when set it indicates that this p?_info structure is being used. The remaining bits are reserved for future use.
p?_id	A 3-byte field that identifies the partition. The field must contain "XGM" to specify that information in this p?_info structure provides a link to the next subdivision.
p?_st	A 68000 format long word that specifies the offset to the beginning of the next subdivision from the beginning of the entire extended GEMDOS partition, in number of physical (512-byte) sectors.
p?_siz	A 68000 format long word that specifies the size of the next subdivision, in number of physical (512-byte) sectors.

The Bad-Sector List

Several terms are used in describing the Bad-Sector List:

BSL:
Bad-Sector List.

Vendor bad sectors:
Bad sectors reported by the vendor of the hard disk device, and those not recoverable by reformatting the device (more detail later). Contrast with *user bad sectors.*

User bad sectors:
Those bad sectors found by a "Markbad" type utility (in HDX) run by the user. These are suspects: it is possible that these will be recoverable by reformatting.

Media BSL:
The bad-sector list placed at the start of the media. This list contains information on both user and vendor bad sectors, in such a way that they can be distinguished. During formatting, the user part of this list is discarded, and the vendor part is kept in memory.

Root sector:
The first sector (sector 0) of a physical device. This contains information about the device: size of device in number of physical (512-byte) sectors: locations, types and sizes of partitions; location and size of the bad-sector list.

Partition:
A GEMDOS "logical" drive, such as C:. There may be from one to four partitions per physical device, and possibly more with the scheme described in the section for Extended GEMDOS Partition.

Partition header:
The *boot sector FATs,* and *root directory* of a GEMDOS partition. It must be contiguous, with no intervening bad sectors.

Boot sector:
The first sector of a *partition.* This gives GEMDOS information about the partition such as its size, the size of its FAT, and how large the root directory is.

FAT:
File Allocation Table. This is where the clusters assigned to a given file in the GEMDOS filesystem are recorded, and also where bad sectors within a GEMDOS partition are marked.

Root directory:
The root (topmost) directory of a GEMDOS partition.

The media BSL is recorded starting at sector "bsl_st" and occupies "bsl_cnt" sectors. "Bsl_st" and "bsl_cnt" are recorded in the root sector, and are described in the section for Hard Disk Partitioning. The size of the BSL is based on the device size, and is fixed at formatting time. This BSL consists of 3-byte entries. The first two entries are special, and are described below. The rest of the entries consist of 3-byte physical sector numbers of the bad sectors. Entries in this list may straddle physical sectors, and a zero-filled entry marks the end of the list, since sector zero can never be bad on a working device.

The first 3-byte entry in the media BSL contains the number of vendor bad sectors recorded. The first byte of the second 3-byte entry is a checksum byte which causes the whole BSL, when added bytewise, to sum to A5 hex. (If this criterion is not met, the whole BSL is assumed to be bad.) The second and third byte of the second 3-byte entry are reserved for future use. The next N entries in the BSL are vendor bad sectors, where N is the number contained in the first entry. The remainder of the BSL is for user bad sectors. The user-bad list is cleared out when the device is reformatted, but is retained during partitioning.

The size of the media BSL is set when the device is formatted: it does not grow. This list is used to remember bad sectors on the media independent of the partitioning scheme. The bad sectors recorded here are also marked in the FAT of each GEMDOS partition, where appropriate. If the partitions are changed, the new FATs will reflect the same bad sectors, relocated appropriately.

Big GEMDOS Partitions

A *big* GEMDOS partition is one whose size is greater than or equal to 16Mb. A big GEMDOS partition is identified by the ASCII characters "BGM" in the 3-byte p?_id field of a p?_info structure in the root sector or an extended root sector. Since a big GEMDOS partition is just like a regular partition, only bigger, it can be made bootable. For information about how to read from or write to big partitions, refer to BIOS FUNCTION - RWABS().

With AHDI 3.00, a partition can be as big as the capacity of a hard disk or a quarter of Gigabyte, whichever is smaller. A big partition is achieved by having bigger logical sectors within the partition. Each time the size of a logical sector is doubled, the maximum size of a partition is doubled. The maximum size of 1/4 Gigabyte is obtained as follows:

Maximum size of a cluster, in number of bytes = 2**14 = 16384
Size of a cluster, in number of logical sectors = 2
Maximum size of a logical sector, in number of bytes = 16384 / 2 = 8192
Maximum size of a partition, in number of logical sectors = 2**15 = 32768
Maximum size of a partition, in number of bytes = 8192 * 32768 = 1/4 Gigabyte

BIOS Parameter Blocks

Each standard partition is represented as a logical drive. The BIOS Parameter Block, called the BPB, provides information about a logical drive. The structure of the BPB has not changed, but the meanings of some fields have.

The 9-word BIOS Parameter Block (BPB) contains this information:

```
BPB ------>   +-----------------------------+      (+0)
              |          recsiz             |
              +-----------------------------+      (+2)
              |          clsiz              |
              +-----------------------------+      (+4)
              |          clsizb             |
              +-----------------------------+      (+6)
              |          rdlen              |
              +-----------------------------+      (+8)
              |          fsiz               |
              +-----------------------------+      (+10)
              |          fatrec             |
              +-----------------------------+      (+12)
              |          datrec             |
              +-----------------------------+      (+14)
              |          numcl              |
              +-----------------------------+      (+16)
              |          bflags             |
              +-----------------------------+      (+18)
```

Figure 4

All words in the structure are in 68000 format.

recsiz	A word that indicates the number of bytes in a logical sector. A logical sector may contain one or more physical (512-byte) sectors.
clsiz	A word that indicates the number of logical sectors in a cluster. The only value supported by GEMDOS is 2.
clsizb	A word that indicates the number of bytes in a cluster. The value is clsiz * recsiz.
rdlen	A word that specifies the size of the root directory, in number of logical sectors. A directory entry uses 32 bytes, so the number of files a root directory can contain is rdlen * recsiz / 32.
fsiz	A word that specifies the size of each File Allocation Table (FAT), in number of logical sectors.
fatrec	A word that specifies the offset to the first sector of the second FAT from the beginning of the logical drive, in number of logical sectors.
datrec	A word that specifies the starting logical sector number of the first data cluster on the logical drive.
numcl	A word that specifies the number of data clusters on the logical drive.
bflags	A word that is used as a bit-vector of flags. Currently only bit 0 is being used; when set it indicates that 16-bit FAT entries (instead of 12-bit ones) are to be used. The remaining bits are reserved.

Boot Sectors

The boot sector is the first logical sector on the logical drive and it occupies one logical sector. When a logical sector contains more than one physical (512-byte) sectors, a boot sector will be bigger than 512 bytes. However, only the first 512 bytes of a boot sector are used, no matter how big the boot sector might be. The layout of the first 512 bytes of a big boot sector is identical to a regular (512-byte) boot sector, the rest of the boot sector is zero-filled.

The first 512 bytes of a boot sector contains the following information:

```
                                  Offset
+--------------------------+      ($0)
|         boot code        |
|         (if any)         |
+--------------------------+      ($8)
|         SERIAL           |
|         24-bit           |
|         volume           |
|         serial *         |
+--------------------------+      ($b)
|    l    bps              |
|    h                     |
+--------------------------+      ($d)
|         spc              |
+--------------------------+      ($e)
|    l    res              |
|    h                     |
+--------------------------+      ($10)
|         nfats            |
+--------------------------+      ($11)
|    l    ndirs            |
|    h                     |
+--------------------------+      ($13)
|    l    nsects           |
|    h                     |
+--------------------------+      ($15)
|         media            |
+--------------------------+      ($16)
|    l    spf              |
|    h                     |
+--------------------------+      ($18)
|    l    nsides           |
|    h                     |
+--------------------------+      ($1c)
|    l    nhid             |
|    h                     |
+--------------------------+      ($1e)
|         boot code        |
|         (if any)         |
|                          |
|                          |
+--------------------------+      ($200)
```

Figure 5

boot code	Byte $0 through $8, and byte $1e through $200 are "don't cares" if the hard drive is not bootable. If the hard drive is bootable, they may contain boot code.
serial *	A 24-bit serial number that is used to help determine if the user has changed cartridges in a removable drive. The serial number is generated randomly and written when a cartridge is partitioned.
bps	An 8086 format word that indicates the size of a logical sector, in number of bytes. One logical sector may contain more than one physical (512-byte) sectors.

spc	A 1-byte field that indicates the size of a cluster, in number of logical sectors. The only value supported by GEMDOS is 2.
res	An 8086 format word that indicates the number of reserved logical sectors at the beginning of the logical drive, including the boot sector. The value of res is usually 1.
nfats	A 1-byte field that indicates the number of File Allocation Tables (FAT) on the logical drive. The value of nfats is usually 2.
ndirs	An 8086 format word that indicates the number of root directory entries on the logical drive.
nsects	An 8086 format word that indicates the total number of logical sectors on the logical drive, including the reserved sectors.
media	A 1-byte field that describes the kind of media the logical drive resides on. For hard disk, the value of media is $f8. The ST BIOS does not use this byte.
spf	An 8086 format word that indicates the size of each FAT, in number of logical sectors.
spt	An 8086 format word that indicates the size of each track, in number of sectors. This field is not applicable to a hard drive.
nsides	An 8086 format word that indicates the number of sides on the media. This field is not applicable to a hard drive.
nhid	An 8086 format word that indicates the number of hidden sectors. This field is not applicable to a hard drive.

The first 512 bytes of an *executable* boot sector *must* word-checksum to the magic number $1234. The last 2 bytes (at offset $1fe) is used for "evening out" checksums. In particular, the Extended BIOS function _Protobt() modifies these 2 bytes. During system initialization, the first 512 bytes of the boot sector from a logical drive are loaded into a buffer. If the checksum is correct, the system JSRs the first byte of the buffer. Since the location of the buffer is indeterminant, any code contained in the boot sector must be position-independent.

When a "Get BPB" call is made, the driver reads the first 512 bytes of the boot sector and examines the prototype BIOS parameter block (BPB). A BPB is constructed from the prototype. If the prototype looks strange (e.g. if critical fields in it are zero) the driver returns zero (as an error indication).

Patchable Variables

In AHDI 3.00, a few variables in the driver are made patchable for the user. These variables do not exist in previous versions of AHDI.PRG or SHDRIVER.SYS. They are placed at the beginning of the driver file (AHDI.PRG or SHDRIVER.SYS).

```
                                          Offset
+-----------------------------------+     ($c)
|   Magic number                    |
+-----------------------------------+     ($e)
|   Version number                  |
+-----------------------------------+     ($12)
|   Ospool size                     |
+-----------------------------------+     ($14)
|   Def_sect_siz                    |
+-----------------------------------+     ($16)
|   * entries in def_ndrv           |
+-----------------------------------+     ($18)
|   1st entry of def_ndrv           |
+-----------------------------------+     ($19)
                 .
                 .
                 .
+-----------------------------------+     ($18 + (n - 1))
|   nth entry of def_ndrv           |
+-----------------------------------+     ($18 + n)
```

Figure 6

Magic number

A value of $f0ad in this word indicates that there are patchable variables in that version of the driver. This magic number $f0ad does not exist in previous versions of AHDI.PRG or SHDRIVER.SYS.

Version number

A 68000 format word that indicates which version of the driver this is. For AHDI 3.00, its value is $0300. This version number does not exist in previous versions of AHDI.PRG or SHDRIVER.SYS.

Ospool size

A 68000 format word that specifies how many "chunks" of memory to add to the OS pool. The default is 128. The size of each chunk is 66 bytes. This number will only be used when the ROM version on your system requires that OS pool be added.

Def_sect_siz

A word that specifies the default logical sector size (in number of bytes) the system will handle. 512 bytes is the smallest number you can specify, which is also the default value of def_sect_siz. The driver will use this number, or the size of the biggest logical sector it could find on all logical drives on the system, whichever is bigger, to be the size of the buffers on the GEMDOS buffer lists.

This is useful when you need to switch cartridges on a removable drive (e.g. MEGAFILE 44) often, and the cartridges are partitioned differently. At boot time, the driver will use this number, or the size of the biggest logical sector on all logical drives, whichever is bigger, to allocate buffers for the GEMDOS buffer lists.

For example, suppose that you boot up the system and the size of the biggest logical sector on all logical drives is 512 bytes. Later, you need something from a cartridge that has a partition whose logical sectors are 1024 bytes big (call it Cartridge A). If the default logical sector size has not been set to be greater than 512, you cannot access this partition on Cartridge A whose logical sectors are 1024 bytes big, because the GEMDOS buffers are not big enough for its logical sectors.

You can reboot with Cartridge A in the drive (so the driver allocates bigger buffers), or you can change this patchable variable so the driver always allocates 1024-byte buffers. You will have to reboot in any case. so the driver can allocate the big enough GEMDOS buffers.

# entries in def_ndrv

A word that specifies the size of the def_ndrv array in number of bytes. The current value of this word is 8, which is the maximum number of ACSI units being supported.

Def_ndrv is an array of bytes that specifies default number of drive letters to be reserved for each ACSI unit. The indices into the array are the physical unit numbers of the ACSI units. This number will only be used if an ACSI unit is a removable hard drive.

This is useful when you need to switch cartridges on a removable drive (e.g. MEGAFILE 44) often, and the cartridges are partitioned differently. At boot time, the driver will use this number, or the number of logical drives on a removable ACSI hard drive, whichever is bigger, and assign that number of drive letters to that particular unit.

For example, suppose that you boot with a cartridge that has two partitions on it (call it Cartridge A) in the removable drive. Later, you need something from another cartridge that has four partitions on it (Cartridge B). If the def_ndrv entry for this removable drive has not been set to be greater than two, you cannot access the last two partitions on Cartridge B, because only two drive letters were reserved for this removable drive.

You can reboot with Cartridge B in the drive (so the driver reserves four drive letters), or you can change this patchable variable so the driver always reserves four drive letters for this physical unit. You will have to reboot in any case, so the new distribution of drive letters is recognized.

1st-nth entry in def_ndrv

A byte that specifies the default number of drive letters to be reserved for unit i, where i = 0, 1, 2, ..., n. The default value for every entry is 1

PUN_PTR

The TOS system variable pun_ptr at $516 points to the following structure:

```
*define     MAXUNITS    16

struct      pun_info {
            WORD        puns;
            BYTE        pun[MAXUNITS];
            LONG        partition_start[MAXUNITS];
            LONG        cookie;
            LONG        cookie_ptr;
            WORD        version_num;
            WORD        max_sect_siz;
            LONG        reserved[16];
}
```

Cookie, cookie_ptr, version_num, max_sect_siz and the reserved fields are the new fields in this structure.

MAXUNITS	A constant that specifies the maximum number of logical drives (including floppy drives A: and B:) supported by the system.
puns	A word that indicates the number of accessible physical units (hard drives) that are connected to the system.
pun	An array of bytes that indicates which physical unit each logical drive resides on. The indices into the array are the logical drive numbers, where 0 is for A:, 1 is for B:, 2 is for C: and so on. Each byte is broken down into:

Bit 0-2	The 3-bit value is the physical ACSI unit number of the unit that the logical drive resides on.
Bit 3-6	These bits are reserved for future use.
Bit 7	When set, this bit indicates that the logical drive does not exist. When not set, it indicates that the logical drive exists.

partition_start	An array of longs that indicates the offset to the beginning of each logical drive from the beginning of the entire physical unit, in number of physical (512-byte) sectors. The indices into the array are the logical drive numbers, where 0 is for A:, 1 is for B:, 2 is for C: and so on.
cookie	A long that indicates more information is following. This cookie does not exist in previous versions of the loaded driver, and so allows programs to determine whether the information they are looking for exists in the verion which is running. The value of the cookie is $41484449, which is 'AHDI' in ASCII.
cookie_ptr	A pointer (which is a long) that points to the cookie. This value is filled in when the driver gets loaded. This allows programs to be sure that they have found the right cookie, not just any random 'AHDI' in RAM.
version_num	A word that indicates which version of the driver is running. For AHDI 3.00, the value of version_num is $0300.
max_sect_siz	A word that indicates the size of the biggest logical sector the system will support. The value is either the size of the biggest logical sector found or the def_sect_siz (as defined in PATCHABLE VARIABLES above), whichever is bigger. This is also the size of the buffers on the GEMDOS buffer lists. If you are writing a program to add buffers to the GEMDOS buffer lists, make sure those buffers are as big as max_sect_siz. This variable is also useful when a program needs to know how big a buffer should be allocated for a logical sector. Allocating max_sect_siz bytes would guarantee the buffer is big enough for any logical sector on all the logical drives.

BIOS Function - RWABS()

Rwabs() is the BIOS call that lets you read or write sectors (logical or physical) on a device. It now takes an extra parameter to address larger hard disks.

```
LONG    rwabs(rwflag, buf, count, recno, dev, lrecno)
WORD    rwflag;
LONG    buf;
WORD    count, recno, dev;
LONG    lrecno;              /* this is the new parameter */
```

rwflag	A bit-vector that indicates the mode of the operation. Bit 0 when set, it's a write operation. when not set, it's a read operation Bit 1 when set, ignores media change when not set, does not ignore media change Bit 2 when set, turns off retry when not set, retries when necessary Bit 3 when set, operates in physical mode when not set, operates in logical mode
buf	A pointer to a buffer to read or write to. In logical mode, the size of the buffer must be at least count * (size of the logical sector). In physical mode, the size of the buffer must be at least count * 512 bytes.
count	In logical mode, this word specifies the number logical sectors to read or write. In physical mode, it specifies the number of physical (512-byte) sectors to read or write.
recno	In logical mode, this word specifies the first logical sector to read from or write to. In physical mode, it specifies the first physical sector to read from or write to. If recno is -1, lrecno will be used instead.
dev	In logical mode, dev specifies the logical drive to read from or write to, and is 0 or 1 for floppy drives A: or B: respectively, and 2+ for hard disks (where 2 is for C:, 3 is for D:, and so on). In physical mode, it specifies the physical unit number of a hard disk, where 2 is for unit 0, 3 is for unit 1, and so on.
lrecno	A long word that specifies the first logical or physical sector to read from or write to. This new parameter is optional and is used only when recno equals -1.

If a logical sector contains more than one physical (512-byte) sectors, Rwabs() will translate the logical sector number to the corresponding physical sector number. Rwabs() will also translate the count of logical sectors to a count of physical sectors. The caller just needs to provide a buffer of appropiate size as specified above.

BIOS Function - GETBPB()

If you plan to use the Getbpb() call, make sure you call the force media change routine before you call Getbpb(). In the driver, there is a flag for each logical drive which tells the system whether medium has changed or not. The flag can have 3 values. A value of 0 means medium has not changed; A value of 1 means medium may have changed; A value of 2 means medium has definitely changed. Each time Getbpb() is called, the flag corresponding to the logical drive in question will be cleared, because the information about that logical drive has been updated. If a medium has changed, and a program calls Getbpb() before GEMDOS has a chance to recognize the medium change, GEMDOS will not see the medium change at all. This is disastrous because GEMDOS will not update its cached information of the logical drive. To make sure GEMDOS will see all possible media changes, you must call the force media change routine to force GEMDOS to recognize a medium change before your program calls Getbpb(). For information about the force media change routine, please refer to FORCING MEDIA CHANGE.

Forcing Media Change

The following is also documented in the Rainbow TOS (TOS 1.4) release notes.

```
*------------------------------------------------------------------------*
*                                                                        *
*     mediach: cause media-change on a logical device.                   *
*                                                                        *
*     USAGE:                                                             *
*           errcode = mediach(devno);    /* returns 1 for error */       *
*           int errcode, devno;                                          *
*                                                                        *
*                                                                        *
*     This procedure causes a media change by installing a new           *
*     handler for the mediach, rwabs, and getbpb vectors; for device     *
*     devno, the mediach handler returns "definitely changed," and       *
*     the rwabs handler returns E_CHNG, until the new getbpb handler     *
*     is called.  The new getbpb handler un-installs the new             *
*     handlers.                                                          *
*                                                                        *
*     After installing the new handlers, this procedure performs a       *
*     disk operation (e.g. open a file) which makes GEMDOS check         *
*     the media-change status of the drive: this will trigger the        *
*     new rwabs, mediach and getbpb handlers to do their things.         *
*                                                                        *
*     RETURNS: 0 for no error, 1 for error (GEMDOS didn't ever do a      *
*           getbpb call; should never happen).                           *
*                                                                        *
*------------------------------------------------------------------------*

           .globl      _mediach
_mediach:
           move.w      4(sp),d0
           move.w      d0,mydev
           add.b       #'A',d0
           move.b      d0,fspec           ; set drive spec for search first

loop:
           clr.l       -(sp)              ; get super mode, leave old ssp
           move.w      #$20,-(sp)         ; and "super" function code on stack
           trap        #1
           addq        #6,sp
           move.l      d0,-(sp)
           move.w      #$20,-(sp)

           move.l      $472,oldgetbpb
           move.l      $47e,oldmediach
           move.l      $476,oldrwabs

           move.l      #newgetbpb,$472
           move.l      #newmediach,$47e
           move.l      #newrwabs,$476
```

```
          ; Fopen a file on that drive

          move.w      #0,-(sp)
          move.l      #fspec,-(sp)
          move.w      #$3d,-(sp)
          trap        #1
          addq        #8,sp

          ; Fclose the handle we just got

          tst.l       d0
          bmi.s       noclose

          move.w      d0,-(sp)
          move.w      #$3e,-(sp)
          trap        #1
          addq        #4,sp

noclose:
          moveq       #0,d7
          cmp.l       #newgetbpb,$472         ; still installed?
          bne.s       done                    ; nope

          moveq       #1,d7                   ; yup! remove & return TRUE
          move.l      oldgetbpb,$472
          move.l      oldmediach,$47e
          move.l      oldrwabs,$476

done:     trap        #1                      ; go back to user mode (use stuff
          addq        #$6,sp                  ; left on stack above)

          move.l      d7,d0
          rts

*-----------------------------------------------------------------------*
*                                                                       *
* new getbpb: if it's our device, uninstall vectors;                    *
*         in any case, call the old getbpb vector (to really            *
*         get it)                                                       *
*                                                                       *
*-----------------------------------------------------------------------*

newgetbpb:
          move.w      mydev,d0
          cmp.w       4(sp),d0
          bne.s       dooldg
          move.l      oldgetbpb,$472   ; it's mine: un-install new vectors
          move.l      oldmediach,$47e
          move.l      oldrwabs,$476
dooldg:   move.l      oldgetbpb,a0     ; continue here whether mine or not:
                                       ; call old.
          jmp         (a0)
```

```
*----------------------------------------------------------------------*
*                                                                      *
* new mediach: if it's our device, return 2; else call old.            *
*                                                                      *
*----------------------------------------------------------------------*

newmediach:
            move.w      mydev,d0
            cmp.w       4(sp),d0
            bne.s       dooldm
            moveq.l     #2,d0           ; it's mine: return 2
                                        ; (definitely changed)
            rts

dooldm:     move.l      oldmediach,a0   ; not mine: call old vector.
            jmp         (a0)


*----------------------------------------------------------------------*
*                                                                      *
*           newrwabs: return E_CHG (-14) if it's my device             *
*                                                                      *
*----------------------------------------------------------------------*

newrwabs:
            move.w      mydev,d0
            cmp.w       $e(sp),d0
            bne.s       dooldr
            moveq.l     #-14,d0
            rts

dooldr:     move.l      oldrwabs.a0
            jmp         (a0)

.data
fspec:      dc.b        "X:\\X",0       ; file to look for (doesn't matter)

.bss
mydev:      ds.w        1
oldgetbpb:  ds.l        1
oldmediach: ds.l        1
oldrwabs:   ds.l        1

*----------------------------------------------------------------------*
*                                                                      *
*           end of mediach                                             *
*                                                                      *
*----------------------------------------------------------------------*
```